1. Assignment guidance

You will produce, in C, a program which fits the following specification:

Maze Game

```
Usage: ./maze <mazefile path>
```

You are creating a basic game, where players navigate through a maze.

The maze will be loaded from a file, the filename for which is passed as a command line argument. Mazes are made up of four characters:

Character	Purpose
'#'	A wall which the player cannot move across
' ' (a space)	A path which the player can move across
'S'	The place where the player starts the maze
'E'	The place where the player exits the maze

A maze has a height and a width, with a maximum of 100 and a minimum of 5.

The height and width do not have to be equal – as long as both are within the accepted range.

Within a maze, each 'row' and 'column' should be the same length – the maze should be a rectangle.

When the game loads, the player will start at the starting point 'S' and can move through the maze using WASD movement:

Key	Direction
W/w	Up
A/a	Left
S/s	Down
D/d	Right

Note: Each input will be separated with a newline character – this is not keypress triggered.

The player can move freely through path spaces (‘ ‘) but cannot move through walls or off the edge of the map. Some helpful prompt should be provided if this is attempted.

The map should not be shown to the player every time they make a move, but they can enter ‘M’/’m’ to view an image of the map, with their current location shown by an ‘X’.

When the user reaches the exit point ‘E’, the game is over and will close. The player should be given some message stating that they have won. There is no ‘lose’ condition.

Mazefile specification

A valid maze:

- Has a single starting point ‘S’
- Has a single exit point ‘E’
- Contains only the start and exit characters, spaces (‘ ‘), walls (‘#’) and newline (‘\n’) characters
- Has every row the same length
- Has every column the same height
- Has a maximum width and height of 100
- Has a minimum width and height of 5
- Does **not** require every row and column to start or end with a ‘#’
- May have a trailing newline at the end of the file (one empty row containing only ‘\n’)

A selection of valid mazes are provided in your starting repository – you should ensure that your code accepts all of these mazes.

Note that file extension is not important – there is no requirement for a mazefile to be stored as a .txt file provided that the contents of the file are valid.

Standard Outputs

To allow some automatic testing of your functionality, we require some of your outputs to have a specific format. To prevent you from being overly restricted, this will only be the **final returned value** of your code rather than any print statements.

Return Codes

Scenario	**Value to be returned by your executable**
Successful running	0
Argument error	1
File error	2
Invalid maze	3
Any other non-successful exit **Note: it is unlikely that you will need to use this code**	100

Maze Printing Function

The maze printing function ('M'/'m') **must** output the maze in the following way:

- No additional spaces added
- Newline before the first row is printed
- Newline after the final row
- If the player's current position overlaps with the starting point, this should display 'X' rather than 'S'

The code required to do this is provided in the template as print_maze() and may be used without referencing me.

Additional Challenge Task – Maze Generator

This is an optional additional task which will involve researching and developing a more complex piece of code – you do not need to complete this section to achieve a good grade.

This task may take longer than the recommended time given above – I recommend only attempting any part of it if you found the original task trivial to complete.

In addition to allowing users to solve mazes, you will create an additional program `mazegen` which allows users to generate a **valid** and **solvable** maze with the specified width and height, to be saved in 'filename'.

For example:

```
./mazeGen maze4.txt 20 45
```

Will save a maze which is 20 x 45 into 'maze4.txt', creating that file if it does not already exist.

Valid maze means that it fits the rules given above, as well as being solvable (there is at least one solution to the maze- it is possible to start at S and exit at E).

There are some existing algorithms which can create mazes, and you should experiment with using these to produce 'quality' mazes which are not trivial to solve, and present some challenge to the player. You should **document** your process of developing the maze creation algorithm, as this will form a part of the assessment.

It is recommended that you keep a log including some maze files generated by each iteration, what you intend to change for the next iteration based on these maze files, and just some general comments about what you think was good or bad about this particular solution.

Some things to consider for each iteration are:

- Did the program produce a variety of designs of maze?
- Did the program produce only valid mazes?
- How did the program perform with larger dimensions (100 x 100 for example)
- What did the program do particularly well?
    - Can you identify what part of the code caused this?
- What did the program do particularly poorly?
    - Can you identify what part of the code caused this?
- What will you try next time?

For this task, you will present your maze generation program to a member of the module team (either in person or through video) and discuss:

- How your program works
- How you iteratively developed it
- The limitations of your solution
- Any improvements you would like to make to it in future

A list of questions will be provided 1 week before presentations/videos are to be completed.

2. Assessment tasks

Produce the C code for a program which solves the tasks detailed above.

You should ensure that your code is:

- Structured sensibly
- Modular
- Well-documented
- Defensive
- Working as intended

You can use the code skeleton you produced in Assignment 1, or a basic skeleton is provided via GitHub Classrooms.

You can use any number of additional header and C files, and a basic makefile has been provided in the original repository to allow compilation – you may edit or replace this if preferred.

You may not use any non-standard C libraries except from unit testing libraries.

You should also use your test script and data from assignment 1 to help you to produce defensive and robust code which fits the specification.

If you did not create a test script, or your test script does not work, then you can manually test your code.

As the test script is not assessed for this work, you may also share your test script with others although you **must not** share any of your c code.

Extension Task

Produce a program able to procedurally generate valid, solvable mazes.

3. General guidance and study support

You should refer to the previous lab exercises and lecture notes to support you. The resources from COMP1711 Procedural Programming may also be useful as these cover the majority of the programming content needed.

4. Assessment criteria and marking process

A full breakdown of the assessment criteria can be found in section 8.

Your code will be tested with a number of different maze files and user inputs containing errors- the exact nature of these errors will not be told to you before marking, so ensure that you validate a wide range of potential user errors. You should use the testscript which you developed for Assignment 1 to check your code.

Your code will be manually checked for code quality.

If you complete the additional challenge task, you will submit your code for plagiarism checking but will present your code to a member of module staff for assessment.